HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Stefan Loser

Shaderbasierende Beleuchtung in der Echtzeitgrafik

2015

Fakultät: MNI

BACHELORARBEIT

Shaderbasierende Beleuchtung in der Echtzeitgrafik

Autor:
Herr Stefan Loser

Studiengang:
Medieninformatik und Interaktives Entertainment

Seminargruppe:
MI11W1-B

Erstprüfer:
Prof. Alexander Marbach

Zweitprüfer:
Diplom-Mathematiker Heiner Schmidt

Einreichung:
Mittweida, 28.01.2015

Faculty of Media

BACHELOR THESIS

Shader-based lighting in realtime graphics

author:
Mr. Stefan Loser

course of studies:
Media Informatics and interactive Entertainment

seminar group:
MI11W1-B

first examiner:
Prof. Alexander Marbach

second examiner:
Graduated Mathematician Heiner Schmidt

submission:
Mittweida, 28.01.2015

Bibliografische Angaben

Nachname, Vorname: Loser, Stefan

Thema der Bachelorarbeit

Shaderbasierende Beleuchtung in der Echtzeitgrafik

Topic of thesis

Shader based Lighting in Realtime graphics

53 Seiten, Hochschule Mittweida, University of Applied Sciences,
Fakultät MNI, Bachelorarbeit, 2011

Inhaltsverzeichnis

Abkürzungsverzeichnis

PC	Personal Computer
AAA	Klassifizierung eines (Spiele-)Projektes
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GLSL	OpenGL Shader Language
HLSL	High Level Shader Language
BRDF	Bidirectional Reflectance Distribution Function
CG	C for Graphics
RGB	Rot Grün Blau
Z-Buffer	Buffer der Tiefeninformationen mit Z als Tiefenachse
UV	Zweidimensionale Texturkoordinaten
GUI	Graphical User Interface
DC	Drawcalls
FPS	Frames per Second

Formelverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorwort

Die Performanceoptimierung ist ein wichtiger Bestandteil in der Spieleentwicklung, da sie die Größe des erreichbaren Publikums beeinflusst. Je geringer die Systemanforderungen des Produktes am Ende der Entwicklung sind, desto mehr Nutzer könnten den Erwerb in Betracht ziehen, weil sie nur wenig zusätzliche oder auch gar keine Computerhardware erwerben müssen, um das Spiel zu spielen. Auf diesem Wege können weitere Kosten für den Konsumenten entstehen. Mit der Folge, dass das Produkt später oder gar nicht gekauft wird. Zusätzliche Kosten entstehen aber auch gleichermaßen für den Entwickler, da dieser seinen Angestellten Computer bereitstellen muss, die die optimalen Systemanforderungen weit überschreiten. Dies liegt daran, dass das entwickelte Spiel in einer Entwicklungsumgebung läuft und diese noch eigene Hardwareleistung benötigt. Weiterhin muss bei einer entsprechenden Zielsetzung im Marketing kommuniziert werden, dass es sich lohnt höhere finanzielle Mittel aufzuwenden, weil die Features, die diese zusätzliche Leistung benötigen, es wert sind. All dies erfordert vom Entwickler ein höheres Budget, das zumeist nur bei großen Unternehmen vorhanden ist.

Die Entwicklung für Spielekonsolen stellt noch eine viel größere Herausforderung im Hinblick auf Performanceoptimierung dar, weil diese eine klare Hardwarebegrenzung besitzt. Diese kann nicht verbessert werden, dennoch wird Innovation im Bereich der Software erwartet. Somit müssen neue Features, die eine ähnliche Qualität auf Spielekonsolen und Computern haben sollen, im höchsten Maße optimiert sein. Oftmals wird dies nicht erreicht und es müssen Einbußen in anderen Bereichen, wie zum Beispiel der Grafik, gemacht werden.

Ein Beispiel für eine überragende Realisierung auf vielen unterschiedlichen Systemen ist „Assassins Creed 4 Black Flag“, welches von Ubisoft entwickelt und veröffentlicht wurde. In diesem sollte nicht nur die Seefahrt, die im dritten Teil bereits möglich war, ein Feature darstellen. Es sollte zudem das Karibische Meer als eigenständiges, riesiges Level komplett begehbar sein. Weiterhin sollte es verschiedene Meeresphänomene und eine Unterwasserwelt mit Rätseln geben. Schiffe sollten sich realistisch anhand des Meeresganges verhalten und dieser sollte abhängig vom Wetter sein. Das Zusammenspiel all dieser Features in Kombination mit dem bekannten Kern des Spiels (als Assassine morden) sollte darüber hinaus ohne Ladezeiten auf Konsolen und dem PC zur Verfügung stehen. Um dies zu realisieren, wurden sieben Studios damit beauftragt, jegliche Features zu realisieren. Nach zwei Jahren

Produktionszeit und mehr als tausend Menschen, die an diesem umfassenden Projekt arbeiteten, wurde es veröffentlicht. Dabei wurde ein Budget von mehreren hundert Millionen Dollar benötigt- ein Aufwand und Risiko gleichermaßen, das die wenigsten Publisher von heute in Kauf nehmen würden und könnten. [1]

[1] Future plc: Assassin's Creed IV is being developed across seven different studios – here's how, URL: http://www.edge-online.com/news/assassins-creed-iv-is-being-developed-across-seven-different-studios-heres-how/#null. letzter Zugriff: 27.01.2015

1 Einleitung

Die Beleuchtung in Videospielen ist in den letzten fünfzehn Jahren, mit der steigenden Leistung der Konsolen und Rechner, zu einem immer wichtigeren Aspekt geworden. Wo am Anfang noch ein schwarzer Punkt unter den Hauptcharakter projiziert wurde (sogenannte Blobshadows), findet heute eine Kombination von dynamischer Beleuchtung und Lightmapbaking statt, sodass jedes Objekt, sogar einzelne Partikel, einen Schatten werfen. Oft sind dies nur Schatten der entsprechenden Objekte in approximierter Form, doch der Grad des Realismus nimmt stetig zu. Es wird noch einige Zeit innerhalb von echtzeitbasierenden Umgebungen dauern bis es tatsächlich möglich ist, realitätsnahe Berechnungen durchführen zu können. Licht basiert auf winzig kleinen Teilchen, genannt Photonen, die mehrfach von den Oberflächen abprallen auf die sie stoßen. Jedes Objekt absorbiert auch einen gewissen Energieanteil. So entsteht indirekte Beleuchtung. Dies ist nur ein Beispiel für die Eigenschaften von Licht. Ein ähnlicher Effekt wird heutzutage mithilfe von ambientem Licht erzeugt. Dabei werden die Schatten durch eine beliebig wählbare Farbe aufgehellt. Dies ist nur ein sehr grundsätzlicher Ansatz, um das reale Verhalten von Licht nachzustellen.

Alles in allem wird Lichtberechnung in Videospielen erzeugt, indem von der entsprechenden Beleuchtungsquelle zu allen Objekten Strahlen, sogenannte Rays, geschossen werden. Da der Abstand zwischen den Strahlen unendlich gering ist, ist für das bloße Auge kein tatsächliches Muster zu erkennen. Je nach der Ausrichtung der Flächennormalen des Objektes wird diese stärker oder schwächer beleuchtet. Dies stellt die diffuse Beleuchtung dar.

Der spekulare Punkt benötigt eine zusätzliche Variable: die Blickrichtung des Spielers. Aus diesen drei Elementen, der Beleuchtungsquellenposition, der Normalenausrichtung und der Blickrichtung des Spielers wird das Phongshading berechnet, welches im Vergleich zu Lambert oder Flatshading sehr aufwändig, aber deutlich präziser und realistischer ist.

Darüber hinaus verwendet die neueste Generation an „AAA“-Spielen, zum Beispiel Kingdom Hearts 3 (produziert mit der Unreal Engine 4), sogenanntes „Physical-based Rendering“. Hierbei werden innerhalb des Shaders die physikalischen Eigenschaften der Oberfläche definiert, damit das Beleuchtungsverhalten realitätsnaher simuliert werden kann. Ein Beispiel dafür ist die Oberflächenrauheit. Wenn die Oberfläche sehr glatt eingestellt wird, reflektiert sie das Licht sehr stark. Die Struktur von Glas ist hierfür das beste Beispiel. Doch wenn sie sehr rau ist, absorbiert sie einen Großteil des Lichtes und sorgt damit auch für weniger „Bouncing Light“, also indirektes Licht.

Produktionen für mobile Endgeräte hingegen haben nach wie vor eine deutlich schwächere Hardwarebasis, erreichen aber mithilfe neuer Techniken eine qualitativ hochwertige Beleuchtung mit dynamischen Schatten.

Viele die genannten Lichtberechnungen werden mithilfe von OpenGL oder DirectX implementiert. Diese sind die Schnittstelle zwischen dem Prozessor (oder auch CPU) und der Grafikkarte (GPU). Sie ermöglichen das Rendern von Objekten im n-dimensionalen Raum. Diese werden meist in Kombination mit der objektorientierten Programmiersprache C++ verwendet und erlauben so die Erstellung von kompletten Spieleengines.

Um die tatsächliche Schattierung eines dreidimensionalen Objektes zu ermöglichen, bietet OpenGL die Shaderprogrammiersprache GLSL und DirectX die Sprache HLSL an. Hierbei werden die meisten Berechnungsschritte schon vor der Shaderkompilierung durchgeführt, um den Shadercode simpel und übersichtlich zu halten.

In der folgenden Arbeit wird erklärt, wie mithilfe des Shaders die komplette Beleuchtungsrechnung durchgeführt werden kann.

2 Die Aufgabenstellung

Das definierte Ziel für dieses Konzept ist es, ein Beleuchtungssystem zu schaffen, welches unabhängig von den internen Funktionen der jeweiligen Technologie arbeitet. Es soll dabei flexible Einstellungsmöglichkeiten innerhalb der Lichtquellen geben und die Übertragung der Werte zum Shader soll leicht und intuitiv für den Anwender sein. Der Shader selbst sollte dennoch leicht zu nutzen und zu erweitern sein. Dabei kann soll er wahlweise komplett dynamische Beleuchtung oder statische ermöglichen.

Es soll, vergleichbar mit der Unity3D Engine oder der Unreal Engine 3, drei verschiedene Lichtquellenarten geben: Punktlichter, Direktionale Lichter und Scheinwerferlichter. Diese sollen folgende Einstellungmöglichkeiten bieten:

Punktlichter	Direktionale Lichter	Scheinwerferlichter
• Lichtart • Lichtintensität • Lichtfarbe • Lichtreichweite • Lichtabschwächung über Distanz • Lichhärte	• Lichtart • Lichtintensität • Lichtfarbe • Lichthärte	• Lichtart • Lichtintensität • Lichtfarbe • Lichtreichweite • Lichtabschwächung über Distanz • Lichthärte • Lichtstrahlwinkel

Tabelle 1: Die Übersicht der Individualisierungmöglichkeiten des entstehenden Shaders.

Über die Einstellungsmöglichkeit der Lichtart soll jederzeit mit einem einzigen Klick die Lichtquellenfunktionalität zu einer beliebigen anderen, zum Beispiel von Punktlicht zu Scheinwerferlicht, geändern werden können.

Die Lichtintensität soll definieren, wie hell das Licht ertrahlen soll, sobald es ein Objekt in der Szene erreicht.

Die Lichtfarbe soll es ermöglichen, die Kolorierung der durch die Lichtquelle angestrahlten Umgebung frei zu ändern. Die Farben der Lichter und der Fläche des Objektes sollen sich dabei mischen.

Die Lichtreichweite soll die maximale Ausbreitung des Lichtes im Raum darstellen, die bei der geringsten Lichtabschwächung über Distanz erreicht werden kann. Bei einer Lichtabschwächung in Höhe von 0 sollen bei einer Reichweite von 50 alle Objekte im Umkreis von 50 Unityeinheiten beleuchtet werden.

Die Lichtabschwächung über die Entfernung, soll den simulierten Zerfall des Lichts über Distanz darstellen. Je höher dieser Wert ist, desto schwächer werden Objekte beleuchtet, die innerhalb der Reichweite des Lichtes sind.

Die Lichthärte soll den Kontrastwert der Beleuchtung zugänglich machen. Da diese oft als Gradient sichtbar wird, soll hier die Anzahl der Farbübergänge modifiziert werden können.

Der Lichtstrahlwinkel soll den Anfangswinkel beschreiben, von dem aus das Licht der Scheinwerferlichtquelle emittiert wird.

Der dazugehörige Shader soll, zusätzlich zu der Darstellung der Lichtquellen, in der Lage sein, eine Vielzahl an Features gleichzeitig zu unterstützen. Zu diesen gehören:

- Diffustexturen
- Diffusfarbe
- Normalentexturen
- Normalentexturanpassung
- Spekulartextur
- Spekularfarbe
- Cubemaps
- Cubemapfarbe

3 Die Geschichte der Beleuchtung in Echtzeitanwendungen

Die Entwicklung der Technologie hinter der Beleuchtung in Videospielen ist in zwei Kategorien einzuteilen: Die Interpolationstechniken und die Beleuchtungsmodelle. Nur das Zusammenspiel dieser beiden Komponenten ermöglicht eine der Realität nachempfundene Schattierung von dreidimensionalen Objekten. Nachfolgend wird auf die Geschichte und den dazugehörigen mathematischen Hintergrund eingegangen.

3.1 Interpolationstechniken

Die einfachste Interpolationstechnik ist das sogenannte Flat Shading, welches zum ersten Mal im Jahre 1967 von der University of Utah durch Romney, Warnock und Watkins beschrieben wurde.[2] Bei diesem wird keine Interpolation zwischen den Polygonfarben vorgenommen. Somit besitzt jede Fläche eine monochromatische Farbe, welche sich durch die Grundfarbe und die Reflektivität der Oberfläche ergibt. Aufgrund der wenigen Parameter, die die Berechnung der finalen Oberflächenfarbe benötigt, ist diese Technik, die auch Konstantes Shading genannt wird, sehr ressourcensparend. Bei gering aufgelösten Objekten bemerkt man eine facettenähnliche Optik (siehe Abbildung 1). [3]

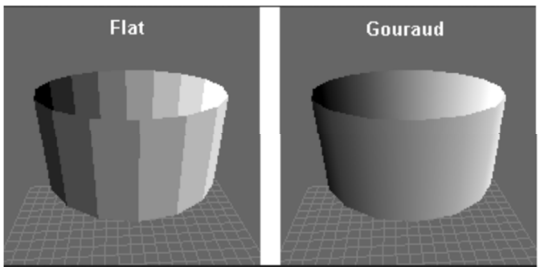


Abbildung 1: Flat und Gouraud Shading im direkten Vergleich.[4]

[2] Henri Gouraud :Continuous Shading of Curved Surfaces, Juni 1971, Seite 1 URL : http://page.mi.fu-berlin.de/block/htw-lehre/wise2012_2013/bel_und_rend/skripte/gouraud1971.pdf. Letzter Zugriff 27.01.2015

[3] Sheilly Padda et al: Different Shading Algorithms for Image Processing, Mai 2014, Seite 2-3, URL: http://www.ijarcsse.com/docs/papers/Volume_4/5_May2014/V4I5-0472.pdf, letzter Zugriff: 27.01.2015

[4] Jon Peddie: The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR and AR, Juli 2013, Seite 59

Im Jahre 1971 veröffentlichte Henry Gouraud das nach ihm benannte „Gouraud Shading“.[5] In diesem wird eine lineare Interpolation zwischen den in den Vertices gespeicherten Farben, anhand deren Normalenausrichtung, vorgenommen. Mithilfe dieser Technik ist trotz vergleichbarer Polygonauflösung ein gleichmäßigerer Farbverlauf innerhalb der Schattierung zu bemerken (siehe Abbildung 1). Dies wird erreicht, indem zuerst die Ausrichtung der Normalen berechnet wird (siehe Abbildung 2). Anhand derer wird die Schattierungsstärke einzeln für jeden Vertice kalkuliert und diese wird entlang der Polygonkanten interpoliert. Im nächsten Schritt wird mithilfe von Abtastlinien die ermittelte Schattierung entlang der Polygonfläche interpoliert (siehe Abbildung 3). Dies eliminiert harte Ränder und ermöglicht so eine kontinuierliche Schattierung über das gesamte, dreidimensionale Objekt. [6]

$$N_v = \frac{\sum_{k=1}^{n} N_k}{\left| \sum_{k=1}^{n} N_k \right|}$$

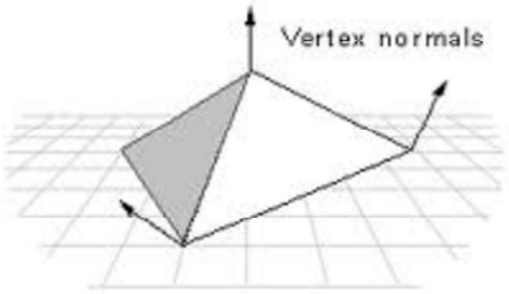
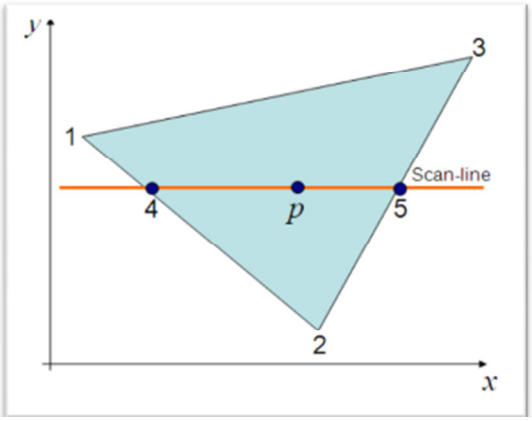


Abbildung 2: Die Berechnung der Vertexnormalen.[7]

[5] Henri Gouraud: Continous Shading of Curved Surfaces, Juni 1971, Seite 1, URL : http://page.mi.fu-berlin.de/block/htw-lehre/wise2012_2013/bel_und_rend/skripte/gouraud1971.pdf. Letzter Zugriff: 27.01.2015

[6, 7, 8] Sheilly Padda et al: Different Shading Algorithms for Image Processing, Mai 2014, Seite 1-2, URL: http://www.ijarcsse.com/docs/papers/Volume_4/5_May2014/V4I5-0472.pdf, letzter Zugriff: 27.01.2015



$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2$$

$$I_5 = \frac{y_5 - y_2}{y_3 - y_2} I_3 + \frac{y_3 - y_5}{y_3 - y_2} I_2$$

$$I_p = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5$$

Abbildung 3: Die Berechnung einer Abtastlinie („Scan-line“) für ein Polygon.[8]

Der Vietnamese Bui Tuong Phong stellte im Jahre 1973 eine weitere Interpolationstechnik vor, das „Phong-Shading“.[9] Im ersten Schritt des dazugehörigen Algorithmus wird, vergleichbar mit dem „Gouraud-Shading“, die Normale jedes Vertices kalkuliert. Danach wird für jedes Pixel, auf dem das dazugehörige Objekt abgebildet wird, eine Normalenausrichtung berechnet und anhand dieser die dazugehörige Schattierungsintensität bestimmt. Am Ende werden die Pixel entsprechend der ermittelten Stärke schattiert. Durch die pixelgenaue Berechnung ist dieser Algorithmus drei mal so aufwändig wie das „Gouraud-Shading“, ermöglicht aber, wie in Abbildung 4 zu sehen ist, deutlich realistischere Ergebnisse. Dies liegt zum Teil an der spekularen Beleuchtung, hier zu sehen an der besonders hellen Schattierung.[10]

Abbildung 4: Von links nach rechts: Flatshading, Gouraudshading und Phongshading im Vergleich.[11]

[9] Jon Peddie: The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR and AR, Juli 2013, Seite 59

[10,11] Sheilly Padda et al: Different Shading Algorithms for Image Processing von ,Mai 2014, Seite 4, URL: http://www.ijarcsse.com/docs/papers/Volume_4/5_May2014/V4I5-0472.pdf, letzter Zugriff: 27.01.2015

Zur Verbesserung der Performance veröffentlichte Jim Blinn im Jahre 1977 das bis heute bekannte „Blinn-Phong“ Shading. In diesem wird, anstatt die Reflektion des Lichtes an der Normalen zu berechnen, ein sogenannter Halfway-Vektor bestimmt. Dieser wird aus dem normalisierten Summand von Lichtrichtung und Blickrichtung bestimmt und im weiteren Verlauf für die Bestimmung des Spekularen Punktes anstelle des Reflektionsvektors genutzt. Das resultierende Ergebnis ist dem Phongshading sehr ähnlich (Vergleich siehe Abbildung 5). [12]

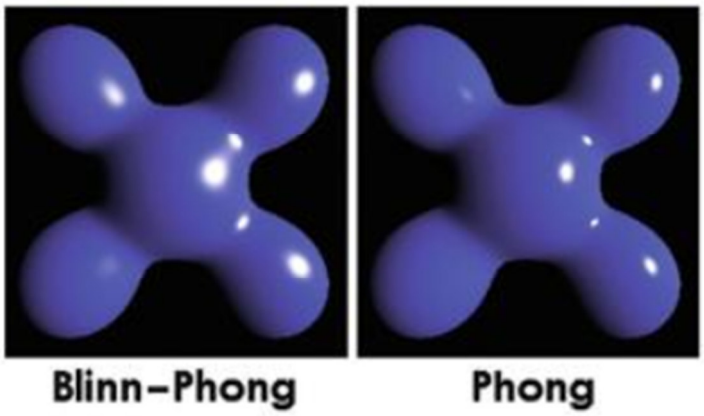


Abbildung 5: Der spekulare Unterschied zwischen Blinn-Phong und Phong im Vergleich. [13]

3.2 Beleuchtungsmodelle

Die „Bidirectional Reflectance Distribution Function“, kurz BRDF, beschreibt wie das Licht mit der Oberfläche auf die es trifft, interagiert . Dabei sind die ausschlaggebenden Faktoren für das entstehende Resultat die eingehende und reflektierte Bestrahlungsstärke und die Strahldichte. Das Verhältnis dieser beiden, in Kombination mit der Farbe, ergibt die finale Beleuchtung.[14] Veröffentlicht wurde dieses Model von F.E. Nicodemus, J.C. Richmond, und J.J. Hsia im Oktober des Jahres 1977. [15]

---

[12] James Blinn: Models of Light Reflection For Computer Synthesized Pictures, Juli 1977, Seite 1, URL: http://research.microsoft.com/pubs/73852/p192-blinn.pdf , letzter Zugriff: 27.01.2015

[13] Jon Peddie: The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR and AR. Juli 2013. Seite 61

[14] Eric Haines et al: Real-Time-Rendering. Februar 2012.Seite 223-224

[15] Nicodemus et al: Geometrical Considerations and Nomenclature for Reflectance. Oktober 1977. Seite 1. URL : http://graphics.stanford.edu/courses/cs448-05-winter/papers/nicodemus-brdf-nist.pdf, letzter Zugriff: 27.01.2015

Die dafür vorhandene Formel beschreibt die Beleuchtung in Abhängigkeit von dem Lichtvektor und dem Sichtvektor. Dies ergibt sich aus dem Quotient zwischen dem Differential von eingehender und reflektierter Strahldichte in Abhängigkeit vom Sichtvektor und dem Differential von eingehender und reflektierter Bestrahlungsstärke in Abhängigkeit vom Beleuchtungsvektor. [16]

$$f(\mathbf{l}, \mathbf{v}) = \frac{dL_o(\mathbf{v})}{dE(\mathbf{l})}$$

Formel 1: Die grundlegende BRDF-Formel nach Nicodemus et al. [17]

Diese Formel ist sehr allgemein und geht davon aus, dass die Materialoberfläche jedes dreidimensionalen Objektes einheitlich ist. Wenn man Flächenlichter außer Betracht lässt, ist die BRDF mit folgender Gleichung beschreibbar[18]:

$$f(\mathbf{l}, \mathbf{v}) = \frac{L_o(\mathbf{v})}{E_L \overline{\cos}\,\theta_i}$$

Formel 2: Die BRDF-Gleichung unter Ausschluss von Flächenlichtern[19]

Hierbei wurde die reflektierte Bestrahlungsstärke anhand des Winkels, ausgehend von der Oberfläche genutzt, um diese genauer zu beschreiben. [20] Möchte man diese Gleichungen für k Lichtquellen generalisieren, ergibt sich:

[16,17,18,19,20] Eric Haines et al: Real-Time-Rendering. Februar 2012.Seite 224-225

$$L_o(\mathbf{v}) = \sum_{k=1}^{n} f(\mathbf{l}_k, \mathbf{v}) \otimes E_{L_k} \overline{\cos}\,\theta_{i_k}$$

Formel 3: Die allgemeine BRDF-Gleichung für k Lichtquellen.[21]

Diese Funktion bescheibt die Vektormultiplikation zwischen dem BRDF und der Bestrahlungsstärke für unendlich viele Lichtquellen k.

Um diese Gleichung auch für realitätsnahe Versuche zu nutzen, wurden zwei grundlegende Bedingungen definiert, die für jede Art physikalisch basierender BRDF zutreffen müssen:

- **Die Erste** Bedingung ist das Einhalten der Heimholtzreziprok. Dieser sagt aus, dass der Eintrittswinkel und Austrittswinkel des Lichtes ausgetauscht werden können, ohne dass sich der Funktionswert ändert.

- **Die Zweite** Bedingung ist der Verbrauch von Energie, der bei dem Auftreffen des Lichtes auf die Fläche stattfindet. In der Physik kann die resultierende Energie nicht der Anfangsenergie entsprechen, da die Oberflächen immer einen Teil absorbieren.

Die hemisphärisch-direktionale Reflexion hat eine direkte Verbindung zu der physikalischen BRDF – sie beschreibt wieviel Energie in einem bestimmten Winkel von der Oberfläche absorbiert wird. Gebildet wird sie durch den Quotient vom Differential zwischen der Ausstrahlung und dem Differential der Bestrahlungsstärke (siehe Abbildung 9). [22]

[21,22,23] Eric Haines et al: Real-Time-Rendering von. Februar 2012.Seite 226-227

$$R(\mathbf{l}) = \frac{M}{E_L \, \overline{\cos}\, \theta_i}$$

Formel 4: Die Gleichung der hemisphärisch-direktionalen Reflexion unter Ausschluss von Flächenlichtern.[23]

Das Lambert'sche BRDF ist eine der Funktionen, die oft in Echtzeitumgebungen genutzt werden. Dabei ist der Wert der hemisphärisch-direktionalen Relfexion konstant. Einzig eine diffuse Farbgebung ohne spekulare Elemente ist bei diesem Beleuchtungsmodel vorhanden. Daher ist diese BRDF mit folgender Gleichung zu definieren[24]:

$$f(\mathbf{l}, \mathbf{v}) = \frac{\mathbf{c}_{\text{diff}}}{\pi}$$

Abbildung 6: Die Gleichung der Lambert'schen BRDF.[25]

Bei dieser Gleichung stammt der invertierte PI-Faktor aus dem integrierten Kosinusfaktor, der sich durch den Einfluss der hemisphärisch-direktionalen Reflexion ergibt. [26]

Es gibt einige weitere BRDF Varianten, zum Beispiel das Phong BRDF oder das Cook-Torrance BRDF. Diese beziehen unterschiedliche Parameter in die Berechnungen mit ein und variieren somit, teils stark, in ihrem Bezug auf physikalische Grundlagen. [27]

[24,25,26] Eric Haines et al: Real-Time-Rendering. Februar 2012.Seite 227-228

4 Lichtquellenarten in der Echtzeitgrafik

Nicht nur die Berechnungen des Lichts wurden in Videospielproduktionen vereinfacht, um Leistung zu sparen. Auch die Arten von Lichtquellen, die jede Spieleengine in der Regel gemein hat, sind identisch und nur sehr gering in der Anzahl. Diese sind Punktlichter, Direktionale Lichter und Scheinwerfer. Mit diesen drei Lichtquellen und der ambienten Beleuchtung wird in Anwendungen wie der Unreal Engine 4, der Unity3D Engine 4 oder CryEngine 3 die komplette Beleuchtung realisiert. Flächenlichter, fachlich „Area Lights“ genannt, stellen eine weitere Art dar, die vermehrt vorzufinden ist.

Bei der Umsetzung der Lichtquellen innerhalb des in dieser Arbeit entwickelten Shaders ist klar zu erkennen, dass die Berechnungen nur sehr wenig Bezug zur realen Physik nehmen und eher funktioneller Natur sind. Da diese Berechnungen bei sehr vielen Objekten innerhalb einer Szene gleichzeitig durchgeführt werden, müssen diese so simpel und optisch qualitativ hochwert wie möglich sein.

4.1 Punktlicht

Punktlichter („Pointlights“) beschreiben Positionen im Raum, die über eine fest definierte Lichtreichweite verfügen und der Distanzabschwächung unterliegen. Diese ist in der Regel innerhalb von Echtzeitumgebungen linear oder quadratisch. Das berechnete Licht breitet sich von der Ursprungsposition allseitig aus, sodass beim bloßen Blick auf die Reaktion einer Fläche auf dieses Licht eine sphärische Form zu erkennen ist (siehe Abbildung 4).Genutzt wird diese Art von Lichtquelle unter anderem, um jegliche Form von Glühlampen darzustellen. [28]

4.2 Scheinwerferlicht

Das Scheinwerferlicht, im englischen Spotlight genannt, stellt eine Position im Raum dar, von der aus ein konischer, über die Distanz zerfallender und direktional ausgerichteter Lichtstrahl ausgesendet wird. Dieser wird für alle Arten von direktionalen Lichtquellen, wie beispielsweise Taschenlampen, verwendet. Durch die konische Form

[28] Eric Haines et al: Real-Time-Rendering. Februar 2012.Seite 218-220

ergibt sich, dass je näher die Ausgangsposition an dem bestrahlten Objekt ist, desto kleiner und heller ist der zu sehende Lichteinfluss. [29]

4.3 Direktionales Licht

Direktionale Lichter, also Lichter die eine Richtung zeigen, wirken im Vergleich zu den anderen Lichtquellenarten positionsunabhängig. Einzig und allein die Ausrichtung bestimmt, aus welchem Winkel alle Objekte in der gesamten Szene beleuchtet werden. Sie sind, funktionell betrachtet, der Ersatz für das Sonnenlicht oder ähnliche unbegrenzte Lichtquellen. Alternativ kann man mit direktionalen Lichtern und sogenannten Lichtmaterialien die Optik des Lichtes individualisieren. Dies wird zum Beispiel in Unterwasserszenarien genutzt, um den Boden des Meeres flächendeckend, zusätzlich zur eigentlichen Gestaltung, mit besonderen Lichtbrechungen zu schattieren.[30]

4.4 Flächenlicht

Flächenlichter sind Scheinwerferlichtern sehr ähnlich. Sie haben einen fest definierten Winkel und werden mit der Distanz schwächer. Der Unterschied zwischen ihnen ist, dass das Flächenlicht keine Ausgangspunkte im Raum hat, sondern eine Ausgangsfläche. Mit dieser Art von Lichtquelle wird umgangen, dass bei weitläufigen strahlenden Flächen mehrere Scheinwerferlichter benötigt werden.

---

[29] Eric Haines et al: Real-Time-Rendering. Februar 2012.Seite 220-222

[30] Eric Haines et al: Real-Time-Rendering. Februar 2012.Seite 218-220

5 Shader in der Unity3D Engine

Die Unity3D Engine bietet dem Programmierer wahlweise die Shadersprachen CG, Shaderlab, GLSL und HLSL. Diese sind über ein entsprechendes Makro anwählbar und es gibt ebenfalls die Möglichkeit, mehrere Sprachen gleichzeitig abzurufen, um verschiedene Features nutzen zu können. Um die Shaderberechnungen herum gibt es ein großes Gerüst, das beispielsweise genutzt wird, um eine bessere Strukturierung zu schaffen und Zugriff auf verschiedene globale Variablen von außerhalb des Shaders zu ermöglichen. Zu diesen gehören die Properties, Tags, Subshader und Pässe.

5.1 Die Properties als Schnittstelle

Der Bereich der Properties, der ein proprietäres Element der Shaderprogrammierung innerhalb von der Unity3D Engine ist, ermöglicht es Variablen mit öffentlichem Zugriff zu definieren. Diese werden denn automatisch mithilfe des internen Materialeditors innerhalb des Inspektors als Teil der Komponente angezeigt. Möglich ist es Vektoren, Floats, Farben, Texturen und Cubemaps anzuzeigen. Für Floats kann man eine direkte Eingabe oder eine Eingabe mit Ober- und Untergrenze ermöglichen. Dies kann über einen Schieberegler eingegeben werden.

Da diese Variablen sowohl instanziert als auch initialisiert werden müssen, ist es notwendig ihnen einen Startwert zu geben. Bei Floats und Vektoren können beliebige Zahlen eingegeben werden und bei Texturen und Cubemaps müssen interne Makros genutzt werden, um bei einem Fehlen keine Artefakte zu erzeugen. Durchschnittliche Diffustexturen und Cubemaps bekommen so eine Farbe zugewiesen, die angezeigt wird. Hierbei kann in Anführungszeichen die Standardfarbe angegeben werden, zum Beispiel „white“ oder „black“. Normalentexturen benötigen hier anstelle von einer Grundfarbe das Stichwort „bump“. Dieses erzeugt eine Normalentextur, die den Normalen des Meshes entspricht. [31]

5.2 Tags

Tags sind ein Teil von Shaderlab und ermöglichen es, verschiedene Eigenschaften von Subshadern und Pässen zu definieren. Beispiele hierfür sind das Individualisieren der Renderqueue oder einen gezielten Beleuchtungsmodus zu definieren.

---

[31] Kenny Lammers: Unity Shader and Effects Cookbook. Juni 2013. Seite 12 -14

5.2.1 Subshader Tags

Es gibt vier Tags, welche man in einem Subshader Tag definieren kann: Queue, RenderType, IgnoreProjector und ForceNoShadowCasting.

In der Queue kann man über Stichwörter wie Geometry, Background, oder alternativ über numerische Eingabe genau definieren, wann das Mesh gerendert wird. Für diese Stichwörter sind innerhalb von Unity Werte hinterlegt. Sie sollen nur die Zugänglichkeit und Lesbarkeit verbessern und Einheitlichkeit ermöglichen. Für Background wird beispielsweise der Wert 1000 eingefügt. Daher sind Ausdrücke wie Transparent+1 ohne Probleme möglich. [32]

Innerhalb des Rendertypes kann definiert werden, ob der Shader volle Opazität, Halbtransparenz oder Transparenz besitzt. Dieser Tag ist Teil des Shaderreplacementsystems und ermöglicht einen gezielten Austausch von Shadern. Somit kann die Kamera über einen entsprechenden Quellcode überprüfen, welche Art von Rendertype das Objekt hat und könnte es nun bei Bedarf ersetzen.

IgnoreProjector ermöglicht lediglich, dass Projektoren kein Bild auf die Oberfläche des Meshes rendern können, dem dieser Shader zugewiesen wurde.

Zuletzt ist ForceNoShadowCasting dafür da, jegliches Erzeugen von Shadowmaps durch das Mesh, welches diesen Shader nutzt, zu verhindern. [33]

5.2.2 Pass Tags

Einer der zwei Pass Tags ist Lightmode. Mithilfe von diesem kann die Beleuchtung genauer definiert werden. Mit dem Stichwort Always verbietet man jegliche Beleuchtung, mit ForwardBase definiert man den dazugehörigen Pass als Grundlage für die Beleuchtung. Dieser würde nur für die erste Lichtquelle eingesetzt werden, die das dazugehörige Objekt trifft. Jede weitere Lichtquelle benötigt einen Pass in welchem ForwardAdd definiert wurde. Dieser wird dann für diese durchgeführt und zu den anderen Pässen addiert.

---

[32] Kenny Lammers: Unity Shader and Effects Cookbook. Juni 2013. Seite 150-151

[33] Unity Technologies: ShaderLab syntax: SubShader Tags. 2014. http://docs.unity3d.com/Manual/SL-SubshaderTags.html, letzter Zugriff: 27.01.2015

Der zweite Tag ist RequireOptions. In diesem können Bedingungen definiert werden, durch welche dieser Pass gerendert wird. Sie müssen extern gesetzt werden. Ein Beispiel hierfür ist SoftVegetation. Ein Pass der mit diesem Tag versehen ist wird nur gerendert, wenn dieses Feature in den Settings aktiviert ist. [34]

5.3 Blending zwischen Pässen

Damit sich verschiedene Pässe nicht gegenseitig in der Hierarchie überschreiben, benötigt man das Blending. Dies ermöglicht eine Kombination der zurückgegebenen Farben mithilfe von mathematischen Formeln. Wenn der erste Pass des Shaders Blending nutzt, könnte dieser mit dem dahinter gerenderten Pixel innerhalb der Szene interagieren und so beispielsweise Halbtransparenz ermöglichen. Alternativ können so Shaderfeatures auf verschiedene Pässe aufgeteilt werden und so wird erreicht, dass jeder Pass des Shader Shadermodel 2.0 benutzt. [35]

Die Grundgleichung, die jedes Mal beim Blending angepasst wird, lautet übersetzt

resultierendesBild = Hintergrundpixel * (vom Anwender defininierter Wert) + PixelDesPasses * (vom Anwender defininierter Wert)

5.3.1 Additives Blending

Mithilfe von additivem Blending, kann man die Farbwerte der RGB-Kanäle zweier Pässe miteinander addieren. Genutzt wird diese Art von Blending um zum Beispiel ein Leuchten zu simulieren. Die Hintergrundfarbe wird mit der entsprechenden Farbe addiert und angezeigt.

---

[34] Unity Technologies: ShaderLab syntax: Pass Tags. 2014. http://docs.unity3d.com/Manual/SL-PassTags.html , letzter Zugriff: 27.01.2015

[35] Unity Technologies: ShaderLab syntax: Blending. 2014. http://docs.unity3d.com/Manual/SL-Blend.html , letzter Zugriff: 27.01.2015

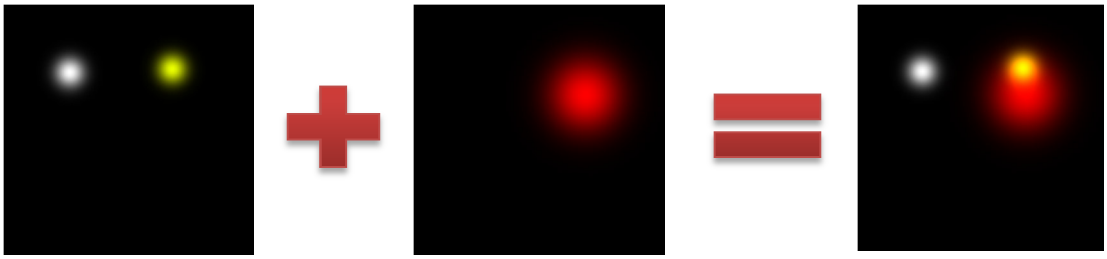


Abbildung 7: Schwarze Pixel werden bei der Addition überschrieben, verschiedenfarbige Pixel mischen sich farblich. In diesem Fall wird aus Rot und Gelb Orange.

Innerhalb des Shaders würde man „Blend One One“ angeben, um diesen Effekt zu generieren. Die folgende Rechnung wird verwendet:

Output = SourceColor * 1 + DestinationColor * 1

Es wird der zuerst gerenderte Pixel mit Eins multipliziert und der neu gerenderte Pixel wird ebenfalls mit Eins multipliziert. Zum Schluss werden beide Ergebnisse addiert. Somit wird effektiv SourceColor + DestinationColor gerechnet. [36]

5.3.2 Multiplikatives Blending

Mithilfe von multiplikativem Blending ist es möglich, die Farbwerte der RGB-Kanäle von zwei Pässen miteinander zu multiplizieren. Dies wird benötigt, um zum Beispiel Elemente zu maskieren und so den Alpha zu beeinflussen.

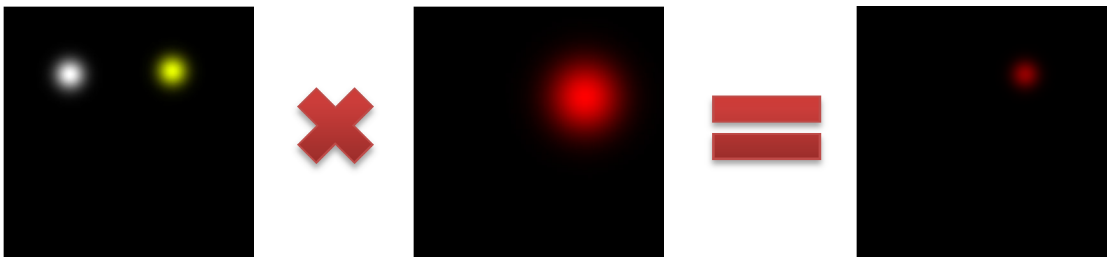


Abbildung 8: Nur Pixel auf beiden Pässen, die ungleich null sind, hellen sich auf und bleiben als nicht-schwarze Grafik bestehen.

[36] Unity Technologies: ShaderLab syntax: Blending. 2014. http://docs.unity3d.com/Manual/SL-Blend.html , letzter Zugriff: 27.01.2015

Die dazugehörige Anweisung im Shader für dieses Blending lautet Blend DstColor Zero. Die daraus resultierende Formel lautet:

Output = SourceColor * DestinationColor + DestinationColor * 0

Somit werden die Pixel des Passes mit denen des Hintergrundpasses multipliziert und hinzuaddiert wird, effektiv Null, weil der zweite Summand mit Null multipliziert wird. [37]

5.3.3 Alphablending

Mithilfe von Alphablending ist es möglich, die Farbwerte der RGB-Kanäle von zwei Pässen, durch die Nutzung des Alphakanals des ersten Passes, zu bearbeiten. Durch das Alphablending wird eine Halbtransparenz innerhalb von Objekten innerhalb der Szene ermöglicht. Es stellt eine der grundlegenden Techniken dar, um jegliche blickdurchlässige Objekte darzustellen.

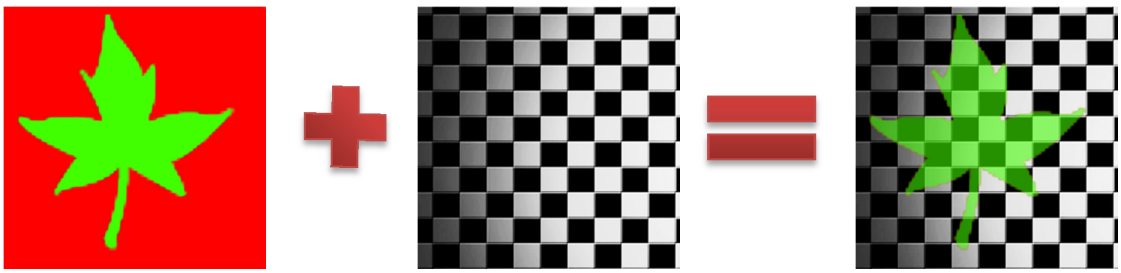


Abbildung 9: Das resultierende Bild sagt aus, dass der Alphawer des Blattes einen Wert zwischen 0 und 255 hatte und man somit bei der Darstellung hindurch blicken kann.

Um dieses im Shader zu bewirken, wird Blend SrcAlpha OneMinusSrcAlpha angegeben. Die nachfolgende Gleichung wird dann verwendet:

Output = SourceColor * SrcAlpha + DestinationColor * OneMinusSrcAlpha

Die RGB-Farben der Pixel des Passes werden mit dessen Alphawerten multipliziert und die RGB-Farben der Hintergrundpixel werden mit den invertierten Alphawerten des Pixelpasses multipliziert.

---

[37], [38] Unity Technologies: ShaderLab syntax: Blending. 2014. http://docs.unity3d.com/Manual/SL-Blend.html, letzter Zugriff: 27.01.2015

Zu beachten ist beim Alphablending, dass das dazugehörige Objekt keine Beleuchtung empfängt und keine Schatten wirft. [38]

5.4 Culling

Durch die Culling-Anweisung ist es möglich zu definieren, welche Faces angezeigt werden und ob dieses Feature genutzt wird. Hierzu wird Culling Front (die nach vorn zeigenden Polygone werden ausgeblendet), Culling Back (die Polygone, deren Ausrichtung entgegengesetzt der Kameranormalen sind, werden ausgeblendet) oder Culling Off (das Ausblenden wird nicht vewendet) genutzt. Die zuletzt verbleibende Option, Culling On, ist der Standardwert und kann nicht manuell angewiesen werden. [39]

5.5 Z-Bufferinformationen schreiben

Mithilfe der „ZWrite“-Anweisung kann man definieren, ob die Koordinaten des Objektes in den Tiefenbuffer, den sogenannten Z-Buffer, geschrieben werden. Die entsprechende Zeile Code hieße dann ZWrite off. Das Gegenstück ZWrite On ist der Standardwert und muss nicht explizit angewiesen werden.

Ein Anwendungsbeispiel ist das Schreiben von Tiefeninformationen bei halbtransparenten Objekten. Wenn deren Koordinaten erfasst werden, ist weder Schattenwurf noch Beleuchtung möglich. Wenn jedoch ein zweiter Pass hinzufügt wird, indem das Mesh mit einem Cutoutshader bearbeitet wird und nur diesen in den Z-Buffer schreibt, sind diese Limitierungen weg und Schatten sind gewährleistet. [40]

5.6 Präprozessorbedingungen

Präprozessorbedingungen sind dafür da, Zeilen zu definieren, die erst bei erfüllter Bedingung durchgeführt werden sollen. Diese werden mit Keywords bestimmt, welche

---

[39, 40] Unity Technologies: ShaderLab syntax: Culling and Depth Testing. 2014. http://docs.unity3d.com/Manual/SL-CullAndDepth.html , letzter Zugriff: 27.01.2015

aktiviert und deaktiviert werden können, je nach dem ob das Feature des Shaders benötigt wird. Dies passiert extern über die C# Programmierung.

Sobald eine solche Bedingung definiert wird, werden vom Präprozessor Shaderpermutationen erzeugt, deren Anzahl exponentiell zur Anzahl der Präprozessorbedingungen wächst. Somit existieren danach Versionen für jede Kombination an Bedingungen, die möglich sind. Wenn während der Laufzeit eine solche verändert wird, wird die Permutation mit einer anderen ausgetauscht. [41]

5.7 Der Vertexshader

Innerhalb des Vertexshaders hat man Zugriff auf alle Eigenschaften des Meshes wie die Normalen- und Tangentenrichtungen, die Vertexpositionen, die Vertexfarben und die UV-Koordinaten. Diese können dann zwischengespeichert werden, um sie mithilfe eines struct-Objects an den Fragmentshader weiter zu reichen. Berechnungen die innerhalb des Vertexshaders durchgeführt werden, werden für jeden Vertex einmal durchgeführt. Dadurch sind Farbverläufe nur auflösungsabhängig erreichbar. Diese Eigenschaft kann auch dafür genutzt werden, Werte die keine hohe Genauigkeit benötigen, kostensparender zu berechnen.Weiterhin können, abhängig von der gewählten Shadersprache, nur bestimmte Features genutzt werden. Beispielsweise unterstützt nur die GLSL das Auslesen von Texturen innerhalb eines Vertexshaders. [42]

5.8 Der Fragmentshader

Innerhalb des Fragmentshaders können Berechnungen pro Pixel durchgeführt werden. Dies erhöht die Präzision von Berechnungen und erlaubt zum Beispiel klare Farbverläufe, vergrößert aber den Aufwand der GPU. Weiterhin hat der Fragmentshader auf keine Eigenschaften des Meshes Zugriff, welche nicht mithilfe des Vertexshaders zwischengespeichert wurden. [43]

---

[41] Unity Technologies: Making multiple shader program variants. 2014. http://docs.unity3d.com/Manual/SL-MultipleProgramVariants.html , letzter Zugriff: 27.01.2015

[42,43] Unity Technologies: Vertex and Fragment Shader Examples. 2014. http://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html , letzter Zugriff: 27.01.2015

6 Die Wahl der Technologie

Für die Umsetzung dieser Aufgabe stehen viele frei benutzbare Spieleengines zur Verfügung: das CryEngine 3 SDK, das Unreal Development Kit und die Unity3D Engine 4.

Aufgrund der guten Zugänglichkeit im Bereich von Shadern und objektorientierter Programmierung ist die Wahl auf die Unity3D Engine gefallen, auch wenn diese die niedrigste Renderingqualität bietet. Weiterhin waren die Beschränkungen der frei zugänglichen Version von Unity ein ausschlaggebender Faktor. Dies ist damit begründet, dass man lediglich auf Forward Rendering zugreifen kann.

Dieses verbietet das Werfen von Schatten von Punkt- und Scheinwerferlichtern im Rahmen der Echtzeitberechnung. Gleichzeitig werden dynamische Objekte auch nur von dynamischen Lichtern angestrahlt. Doch diese sind leistungsaufwändig und werden deshalb in der Regel nicht verwendet. Eine Technik namens Lightmap baking, welche es ermöglicht aus der Beleuchtung eine Textur zu erstellen und diese dem Shader zu übergeben, wird alternativ verwendet. Diese Technologie ermöglicht eine hohe Kostenersparnis, verbietet aber jeglichen Lichteinfluss bei dynamischen Objekten in der Szene. Beispiele dafür sind der Spielercharakter, sich bewegende Bäume oder auch künstliche Intelligenzen.

Durch die shaderbasierende Beleuchtung wird ein System innerhalb der frei verfügbaren Unity3D Engine geschaffen, welches zusätzliche Lichteinflüsse auf beliebige, dynamische und statische Objekte ermöglicht.

Ein weiterer Grund für die Wahl von Unity als Entwicklungsumgebung war die Option, Shaderpermutationen explizit nutzen zu können. Diese offerieren umfangreiche Möglichkeiten der Individualisierung des Shaders je nach benötigten Features. Da Unity ein hohes Maß an Flexibilität in der Erstellung der graphischen Benutzeroberfläche bietet, ergibt sich ein vorteilhaftes Zusammenspiel. Shaderfeatures können nicht nur ausgeblendet, sondern auch nicht kompiliert werden, was somit die Performanz des Systems deutlich verbessert und eine aufgeräumtere Benutzeroberfläche schafft.

Ein umfangreicher Shader hat den Nachteil, das Shadermodell 3.0 nutzen zu müssen. Dennoch ist somit die Möglichkeit gegeben, komplexe Szenen oder ganze Spiele mit nur einem Shader zu erstellen, was die Nutzerfreundlichkeit erhöht und die Leistungsanforderungen senkt.

7 Das Konzept der Realisierung

Das System für die shaderbasierende Beleuchtung wird strikt nach dem „Modell View Controller“ – Prinzip durchgeführt. Dieses besagt, dass das Modell die logischen Berechnungen beinhaltet und Daten speichert. Die View ist rein zu optischen Zwecken vorhanden und dient als Schnittstelle zwischen Nutzer und System. Der Controller wertet die Nutzereingaben aus und liest die Daten der View und übergibt sie an das Modell.

Model	View	Controller
• Shader • Lichtquelle	• LichtquellenGUI • ShaderGUI • ManagerGUI	• Lichtquellenmanager

Tabelle 2: Die Klassenübersicht des entstehenden Beleuchtungssystems.

Der Shader beinhaltet alle Beleuchtungsberechnungen für jede Art von Lichtquelle. Alle Werte die direkt mit den Eigenschaften des Lichtes (wie zum Beispiel Farbe oder Helligkeit) zutun haben, werden vom Lichtquellenmanager gesetzt, welcher diese aus den einzelnen Lichtquellen in der Szene bezieht. Für jede Lichtqelle wird der Shader einen Pass besitzen. Alle Lichtquellenpässe werden additiv ineinander geblendet und zum Schluss wird der Pass mit allen Texturen multiplikativ hineingeblendet.

Die Lichtquelle ist ein Container für alle Eigenschaften, die jede beliebige Lichtart annehmen kann. Diese können über Getter- und Setter-Methoden abgerufen und bearbeitet werden. Darüber hinaus ist das Skript in der Lage, je nach ausgewählter Art des Lichtes eine entsprechende Visualisierung der entsprechenden Lichteigenschaften zu zeichnen. Beim direktionalen Licht wird ein Pfeil mit der Lichtrichtung sichtbar, beim Punktlicht wird durch eine Sphäre der maximale Lichtradius deutlich und beim Scheinwerferlicht ergibt sich aus vier Pfeilen eine konische Form, welche Richtung und Reichweite der Lichtquelle exakt anzeigt.

Die GUI der Lichtquellen ermöglicht alle verfügbaren Einstellungen, die sonst im Shader gemacht werden müssten, direkt in der Lichtquelle einzustellen. Damit können die Auswirkungen der Lichtquellen klar getrennt voneinander betrachtet werden. Dieses System soll mehr Transparenz bieten und ist an das native System von Unity angelehnt. Eingabeelemente sind Checkboxen für das Auswählen der Art der Lichtquelle, Farbwähler für die Lichtfarbe und numerische Felder für die freie Eingabe

von Werten wie Lichtreicheweite oder Lichtabschwächung. Beim Aktivieren eines Checkboxen (zum Beispiel „Punktlicht“) wird die graphische Oberfläche dynamisch aktualisiert und zeigt nur noch Eigenschaften an, die auch vom Punktlicht unterstützt werden. Alle anderen Checkboxen werden deaktiviert.

Die graphische Oberfläche des Shaders ist dafür da, jede Art von optischer Individualisierung mithilfe von Texturen zu ermöglichen. Diese können, vergleichbar mit dem bereits eingebauten System innerhalb von Unity, per Drag and Drop hinzugefügt werden. Zusätzlich sind Parameter zum Einfärben von Texturen und zum Kontrollieren der Normalenwerte vorhanden.

Über die Benutzeroberfläche des Managerskripts können die Lichtquellen, welche in die Berechnungen der Shader einbezogen werden sollen, in eine Liste eingefügt werden. So können während der Laufzeit so viele Lichtquellen hinzugefügt werden, wie der entsprechende Shader Pässe hat. Grundsätzlich wird dieses System so aufgebaut, dass unendlich viele Lichtquellen unterstützt werden, um den verschiedensten Anforderungen zu entsprechen. Falls sich die Anzahl der Objekte in der Szene verändert und somit die Anzahl der Materialien, die den unterstützten Beleuchtungsshader nutzen, können über einen Updatebutton alle Shaderwerte innerhalb der neuen Objekte geupdatet werden.

Der Lichtquellenmanager ist für die Werteübertragung der Variablen von den Lichtquellen zu entsprechenden Shadern verantwortlich. Falls eine Lichtquelle entfernt wird, müssen die Werte der Pässe auf ihre Standardwerte zurückgesetzt werden. Darüber hinaus aktiviert und deaktiviert er Präprozessorbedingungen, um die Effizienz der Shader zu steigern.

8 Das Lichtquellenhilfssystem

Bei der Nutzung dieses Beleuchtungssystems wäre es möglich, jegliche Eingaben rein über den Shader durchzuführen. Im Hinblick auf andere Engines und deren Beleuchtungssysteme stellt ist schnell festzustellen, dass dies sehr unüblich wäre. Darüber hinaus ist im Sinne der Benutzerfreundlichkeit ein optisches System sehr zuträglich. Es ermöglicht eine optische Repräsentation verschiedenster Lichteigenschaften in einer Szene zu platzieren und vereinfacht so beim Anlegen und späterer Nachbearbeitung einen effizienteren und einsteigerfreundlicheren Workflow. Diese Werte müssen dann zum Shader übertragen werden. Dies ist ein Prozess, der, wenn er in jedem Frame durchgeführt wird und somit bei einer Änderung der Position oder Rotation oder einer anderen Eigenschaft sofortige optische Anpassung zeigt, sehr ressourcenaufwändig aber gleichzeitig umgänglich ist. Da am Ende der Bearbeitung diese Werteübertragung abgeschalten werden kann, wird die Performanz während der Laufzeit nicht beeinträchtigt. Somit wird ein intuitiv nutzbares und gleichzeitig performantes System hinzugefügt, welches keine Nachteile für den endgültigen Shader oder die Anwendung hat.

8.1 Das Lichtquellenskript als Visualisierungshilfe und Eigenschaftencontainer

Die Lichtquelle hat primär zwei Aufgaben: die Verbesserung der Anwenderfreundlichkeit und das Speichern der gewünschten Werte in einem eigenen, im System unabhängigen Container. Darüber hinaus wird Zugriff auf alle gespeicherten Werte ermöglicht, welche teilweise weiterhin bearbeitet werden können, um die Rechenschritte innerhalb des Shaders zu minimieren.

8.1.1 Die funktionalen Elemente des Lichtquellenskripts

Viele der Werte, die das Skript über die graphische Oberfläche erhält, werden unverändert gespeichert und sind dann jederzeit abrufbar. Dies sind zum Beispiel die Lichtfarbe oder der Lichtradius. Doch es gibt einige Variablen, deren Inhalt noch bearbeitet wird, um die Rechenschritte innerhalb des Shaders zu verringern. Ein Beispiel dafür ist, die Position des Lichtes innerhalb des Weltkoordinatensystems anstelle vom lokalen Koordinatensystem zu speichern. Somit spart man eine Matrizenmultiplikation innerhalb des Shaders. Für jeden Wert gibt es eine get()- und set() –Methode, um das Verändern der privaten Variablen zu ermöglichen. Aufgerufen werden diese von der graphischen Oberfläche, um Werte abzurufen und anzuzeigen und dem Managerskript, um die Werte im Shader zu setzen.

Eine weitere Funktion des Lichtquellenskripts ist es, abhängig von der Art der ausgewählten Lichtquelle verschiedene primitive Formen zu zeichnen, um die jeweiligen speziellen Eigenschaften des Lichtes besser zu visualisieren. Dies wird mithilfe der Gizmosbibliothek innerhalb von Unity durchgeführt und unabhängig von dem Gameobjekt ausgeführt, welches dieses Skript als Komponente trägt.

Zuletzt besitzt die Lichtquelle zur Erleichtung des Worflows die Funktion, dass sie, sofern noch kein Managerskript existiert, ein neues Objekt mit dieser Komponente anlegt. Somit wird ein weiterer manueller Schritt eingespart.

8.1.2 Die graphische Benutzeroberfläche des Lichtquellenskripts

Die graphische Benutzeroberfläche besteht aus sechs permanenten Grundelementen und abhängig von der derzeitigen Auswahl der Lichtquellenart werden dynamisch mehrere erforderliche Parameter hinzugefügt. Zu den beständigen Anzeigen gehören drei Checkbuttons mit deren Hilfe man die Art der Lichtquelle bestimmt. Diese basieren auf boolschen Werten und beschreiben nur wahr oder falsch. Bedingung für dieses System ist es, dass nur einer der Buttons auf wahr gesetzt werden kann. Wann immer also auf einen Button geklickt wird, werden alle anderen auf false gesetzt. Dies vermeidet Laufzeitfehler des Shaders oder unerwünschte Effekte.

Die restlichen Elemente sind die Lichtintensität, Lichtfarbe und Lichthärte. Dies sind alles Eigenschaften, die bei der Kalkulation jeglicher Art von Beleuchtung benötigt werden. Dabei können die Lichtintensität und die Lichthärte mit einem Floatfeld, also einem Textfeld in das Floats eingetragen werden können, beeinflusst werden. Die Lichtfarbe kann mithilfe eines Farbwählers neu gesetzt werden.

Bei der Anwahl des direktionalen Lichts werden keine weiteren graphischen Elemente hinzugefügt. Die Anzeige bleibt unverändert.

Sobald der Boolean des Punktlichts auf wahr gesetzt wird, werden zwei weitere Floatfelder dynamisch hinzugefügt. Mit diesen kann man den Zerfallsquotienten und die Reichweite des Lichtes einstellen.

Wenn das Scheinwerferlicht ausgewählt wird, werden die Floatfelder für die Reichweite, den Winkel und den Zerfallsquotienten des Lichtes hinzugefügt.

8.2 Das Lichtmanagerskript als Kontrollelement

Das Lichtmanagerskript dient als Schnittstelle zwischen dem Shader und der Lichtquelle. Es befindet sich auf einem eigenständigen Gameobjekt innerhalb der Szene und wird automatisch angelegt, sobald es mindestens ein Lichtquellenskript als Komponente zu einem Gameobjekt hinzugefügt wird.

8.2.1 Die funktionalen Elemente des Lichtquellenmanagerskripts

Das Lichtquellenmanagerskript ist das Kontrollelement dieses Systems und besitzt drei Hauptaufgaben: Das Finden von Materialien innerhalb der Szene mit dem gewünschten Shader, das Übertragen der Werte von den Lichtquellen zu den Materialien in Echtzeit und das Aktivieren bzw. Deaktieren von Präprozessorbedingungen innerhalb aller Zielshader.

Das Durchsuchen der Szene nach Shadern passiert über die interne FindObjectOfTypeAll()-Methode. Mithilfe dieser werden alle Renderer erfasst und nun entsprechend der Anforderungen mithilfe von if-Abfragen aussortiert oder in einer Liste deponiert. Als Auswahlkriterium wurde der Name des Shaders gewählt. Bei einer Erweiterung des Systems und einer Einführung eines Präfixsystems für die Namen der Shader, ist diese Art der Sortierung weiterhin ohne Probleme anwendbar.

Um die Werte des Shaders ständig aktuell zu halten, wird die Update()-Methode genutzt. Diese wird in jedem Frame neu aufgerufen und ermöglicht dadurch schnelles Feedback des Shaders bei Änderung der Lichtquelleneigenschaften. Zum Setzen der Eigenschaften werden die SetFloat() und SetVector()-Methoden genutzt, die die Unity3D Engine anbietet. Die Werte von den jeweiligen Lichtquellen werden direkt über die entsprechende get()-Methode referenziert und dann gesetzt. Da dieses System alle Materialien und Lichtquellen in Listen speichert und die Werte mithilfe von for-Schleifen setzt, ist es vorbereitet für eine unendlich Anzahl an Lichtquellen mit einer unbegrenzten Anzahl an Materialien.

Um die Präprozessorbedingungen zu aktieren und deaktivieren, werden die proprietären Methoden EnableKeyword() und DisableKeyword() genutzt. Diese ermöglichen, dass eine andere Permutation des Shaders mit der derzeitigen ausgetauscht wird und somit nur die gewünschten Features kompiliert und gerendert werden. Im ersten Schritt werden anhand der Anzahl der derzeitigen Lichtquellen innerhalb der Liste alle anderen deaktiviert. Im zweiten Schritt wird überprüft welche Art von Lichtquelle zur Laufzeit gerendert wird und deaktiviert die Berechnungen und

Speicherallokierungen von Variablen, die für die ausgewählte Lichtart nicht benötigt werden.

8.2.2 Die graphischen Elemente des Lichtquellenmanagerskripts

Es gibt zwei Möglichkeiten des Benutzers mit dem Managerskript zu interagieren: über ein Liste mit Lichtquellen, die per Hand zugewiesen werden müssen oder über einen Button, mit dessen Hilfe die Szene erneut nach neuen Objekten durchsucht wird, um die Referenzen zu Materialien herzustellen, die den Zielshader besitzen.

Jedes Managerskript besitzt eine Liste für alle Lichtquellen, die von dem Skript genutzt werden sollen. Diese müssen händisch hinzugefügt werden, nachdem sie erstellt wurden. Dass alle Lichtquellen sich nicht automatisch beim Managerskript anmelden, hat den Vorteil, dass die Verwaltung verschiedener Lichtquellen in Zusammenhang mit bestimmten Shadern vereinfacht wird. Ein Beispiel hierfür: ein Lichtmanager, der für direktionale Lichter zuständig ist, sollte komplett alle Shader kennen, die in der Szene vorhanden sind und diese mit Informationen versorgen. Ein örtlich begrenztes Pointlight hingegen sollte nur sehr wenige Shader kennen. Dies bedarf einem höheren Planungsaufwand bei der Erstellung der Szene, ermöglicht im Nachbearbeitungsprozess jedoch bessere Übersicht und während der Bearbeitung eine bessere Performanz.

Der Button zum Aktualisieren der Materialreferenzen wird gebraucht, da das Durchsuchen jedes Objektes innerhalb der Szene nach dem dazugehörigen Renderer sehr kostenaufwändig ist. Wenn dieser Prozess in Zuge der Updatemethode oder auch in der FixedUpdatemethode durchgeführt werden würde, würde das die Performance des Projektes, abhängig von dessen Größe, deutlich schmälern. Mithilfe dieses Buttons kann die Szene manuell durchsucht und die Beleuchtung neu berechnet werden, sobald die Szenerie vergrößert wird.

9 Der Beleuchtungsshader

Der Beleuchtungsshader ist in diesem System Teil des Modells, da er das Gros der Berechnungen durchführt und entsprechende Methoden bereitstellt. Aufgrund der vielfältigen Anforderungen an diesen Shader sind seine Funktionalitäten in verschiedene Bereiche aufgeteilt.

Innerhalb der Properties werden alle für den User verfügbaren Individualisierungsoptionen ermöglicht. Diese werden durch ein internes Skript im Inspektor als graphische Oberfläche dargestellt und ermöglichen so die Interaktion.

Weiterhin wird für jede Lichtquelle ein Pass benötigt, welcher dann beleuchtungsspezifische Berechnungen durchführt. Dieser Aufbau sichert eine einfache Erweiterbarkeit in komplexen Szenen. Zuletzt gibt es einen Pass, der nur für die Berechung der Texturen, welche durch den Nutzer hinzugefügt wurden, zuständig ist. Dieser bietet viele individualisierbare Eigenschaften, welcher den Shader sehr flexibel für alle Arten von dreidimensionalen Umgebungsgegenständen macht.

9.1 Die Properties des Shaders

Innerhalb des Shaders gibt es für jeden einzelnen Lichtquellenpass zwei Vektorproperties für die Lichtquellenposition und die Lichtrichtung, fünf Floatproperties für Lichtreichweite, die Lichthärte, den Öffnungswinkel, die Intensität des Lichtes und die Zerfallskonstante und eine Farbproperty für die Lichtfarbe. Diese werden vom Lichtquellenmanager gesetzt und sind durch das Makro [HideInInspector] nicht auf der graphischen Oberfläche sichtbar. Dadurch wird die Oberfläche entschlackt und die Anwenderfreundlichkeit erhöht.

Darüber hinaus existieren Texturproperties für die Diffus-, Cubemap-, Spekular- und Normalenindividualisierung, wobei die ersten drei Genannten zusätzlich noch die Möglichkeit einer Farbkolorierung über eine Farbproperty besitzen. Die Normalentextur kann darüber hinaus über einen Multiplikator abgeschwächt oder verstärkt werden, um die Reaktion der Oberfläche auf das Licht zu verändern.

9.2 Der Beleuchtungspass im Detail

Dieser Pass des Shaders beinhaltet jegliche Berechnungen, die mit der Beleuchtung des Objektes zutun haben. Im Vergleich zu Unitys internem System, benötigt dieses mehr Schritte um die vollständige und korrekte Beleuchtung zu gewährleisten. Der Vorteil dabei ist, dass es die Möglichkeit besitzt, jederzeit erweitert und individualisiert

werden zu können. Dieser Renderpass besitzt verschiedene Elemente, die gemeinsam eine überschaubare und dennoch komplexe Struktur ermöglichen.

9.2.1 Das Blending der Beleuchtungspässe

Das Ineinanderblenden der Pässe ist ein Kernelement des Shaders und ermöglicht eine nutzerfreundliche Strukturierung und eine Berechnungsaufteilung, die eine generische und unendlich erweiterbare Grundform zulässt. Durch einfaches Hinzufügen eines weiteren Passes mit den korrekten Blendingeinstellungen kann eine weitere Lichtquelle vom Shader bearbeitet und angezeigt werden.

Zum Erstellen einer Beleuchtungstextur, die ein gutes Ergebnis ermöglicht, sind zwei Schritte zu beachten. Im ersten Schritt wird jeder Hintergrundpixel vom ersten Beleuchtungspass überschrieben. Das bedeutet, dass dieser Pass kein Blending nutzt und die erste Lichtquelle abbilden kann.

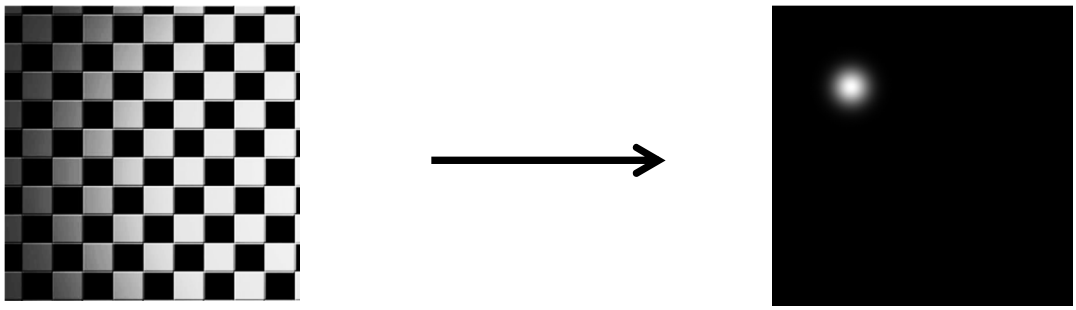


Abbildung 10: In diesem Beispiel wird jeder unbeleuchtete Teil komplett schwarz, Faktoren wie ambient Lighting werden der Eindeutigkeit halber nicht einbezogen.

Nun müssen im zweiten Schritt alle restlichen Beleuchtungspässe zu dieser addiert werden, um die entgültige Textur zu bilden.

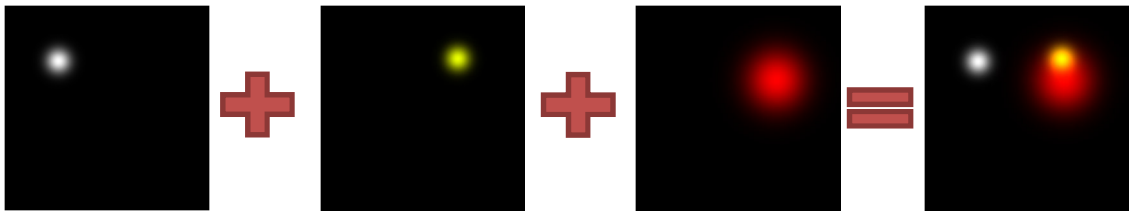


Abbildung 11: Die Addition dreier exemplarischer Beleuchtungstexturen.

Diese wird dann im Texturpass durch weiteres Blending angepasst und bildet die endgültige Oberflächenschattierung.

9.2.2 Präprozessorbedingungen im Beleuchtungspass

Da jeder Pass die Berechnungen für jede Lichtquelle beinhaltet, um den Workflow so einfach und flexibel wie möglich zu halten, musste ein System entwickelt werden, welches Pässe anhand der benötigten Anforderungen anpasst. Indem Präprozessorbedingungen genutzt werden, können die Berechnungen pro Pass, die durchgeführt werden, auf ein Minimum reduziert werden.

Innerhalb des Shaders werden Variablen, die von allen Lichtquellenarten benötigt werden, sowohl instanziert als auch initialisiert. Dies ermöglicht, sofern im jeweiligen Pass keine Berechnungen benötigt werden, dass trotzdem am Ende des Passes ein darstellbarer Wert vorhanden ist. Zusätzlich werden lichtberechnungsspezifische Variablen, wie der Austrittswinkel des Scheinwerferlichts, nur global instanziert und später im Shader initialisiert, wenn die entsprechende Bedingung aktiviert wurde.

Die konkreten Bedingungen Is_DirectionalLight, Is_SpotLight und Is_PointLight befinden sich in jedem Pass und besitzen am Ende noch zusätzlich eine Nummer die definiert, im wievielten Beleuchtungspass sie vorhanden sind. Sie umschließen die jeweiligen Berechnungen für den diffusen und spekularen Anteil der Zielbeleuchtung, genauso wie die Distanzabschwächung. Diese ist nur im Punktlicht und Scheinwerferlicht vorhanden. Somit wird nur eine Shaderpermutation abgerufen, die die Abschwächung unterstützt, sofern eine der beiden Bedingungen erfüllt wurde.

9.2.3 Der Vertexshader des Beleuchtungspasses

Im Vertexshader des Beleuchtungspasses werden, um die Normalentextur zu ermöglichen, die Vektoren für die Normale, Tangente und Binormale berechnet. Weiterhin wird der Sichtvektor berechnet, um spekulare Glanzpunkte erzeugen zu können. Zuletzt werden die Texturkoordinaten unverändert zwischengespeichert und die neue Objektposition wird durch eine Multiplikation mit der Model-View-Projection-Matrix ermittelt.

9.2.4 Der Fragmentshader des Beleuchtungspasses

Im Fragmentshader wird ein Großteil der Berechnungen durchgeführt, da die Beleuchtung sehr gleichmäßig und verticeunabhängig sein muss.

Zuerst wird die Normalentextur gelesen und mithilfe der im Vertexshader berechneten Werte in das Weltkoordinatensystem übertragen. Diese ersetzt nun in den folgenden Rechnungen die Normalenrichtung des dazugehörigen dreidimensionalen Objektes. Eine zweite Textur für den spekularen Glanzpunkt wird noch darüber hinaus ausgelesen. Das sind die einzigen zwei lichtquellenunabhängigen Berechnungen. Jede Lichtquellenart berechnet drei Werte: die Lichtrichtung, die diffuse Schattierung und den spekularen Glanzpunkt. Daraus wird dann das Phongshading gebildet.

Für das Punktlicht und das Scheinwerferlicht wird die Lichtrichtung aus der Differenz zwischen Lichtquellenposition und den Objektkoordinaten gebildet. Da das direktionale Licht entfernungsunabhängig ist, benötigt dieses nur den im Skript berechneten Richtungsvektor, der sich an der Normale des Objekttransforms orientiert. Damit zeigt das Licht immer in Richtung des Objektes.

Für diffuse Schattierung benötigen Punkt- und Scheinwerferlicht die Distanzabschwächung. Diese wird errechnet, indem zuerst die invertierte Distanz der Lichtquelle zum Objekt berechnet wird und dann durch die gewünschte Reichweite dividiert wird. Um negative Zahlen vorzubeugen, werden sowohl Distanz als auch Reichweite quadriert. Um den dabei entstehenden Gradienten besser zu kontrollieren, werden zuerst Zahlen die kleiner als Null sind herausgefiltert. Das Ergebnis kann dann mit einem beliebigen, in der Lichtquelle einstellbaren Wert der Lichtzerfallsquote, bearbeitet werden. Dieses wird als Exponent genutzt. Um seltsame Ergebnisse zu vermeiden, werden Werte kleiner als Null und größer als Eins herausgefiltert.

Der zweite Faktor der diffusen Schattierung ist der Beleuchtungsgradient. Dessen Berechnung ist beim direktionalen Licht und Punktlicht ähnlich. Hierbei wird mithilfe des Skalarprodukts der Winkel zwischen der Normalenrichtung und der invertierten Lichtrichtung bestimmt. Der entstehende Wert kann mit dem in der Lichtquelle stehenden Parameter Lichthärte, Lichtfarbe und Lichtintensität beeinflusst werden.

Für das Scheinwerferlicht wird zuerst die Objektausrichtung in Weltkoordinaten umgerechnet und dann der Winkel zwischen dieser und der Lichtrichtung bestimmt. Danach wird der eingegebene Winkel als Basis für eine Interpolation zwischen 0 und 1 genutzt, um so eine normalisierte Winkelgröße zu erhalten. Falls nun der Winkel zwischen Objektausrichtung und der Lichtrichtung größer als der eigegebene Winkel ist, wird eine Beleuchtung gezeichnet, die Distanzabschwächung, Diffus- und Spekularbeleuchtung hat. Diese wird dann mit dem invertierten Quotienten zwischen eingegebem Beleuchtungswinkel und dem Winkel zwischen Objektausrichtung und der Lichtrichtung multipliziert, um die konische Form zu erhalten. Wenn die oben genannte Bedingung nicht eingehalten wird, wird schwarz gerendert.

Der spekulare Glanzpunkt wird bei allen drei Lichtquellenarten gleich ermittelt. Die invertierte Lichtrichtung wird an der Normale reflektiert und der Winkel zwischen dem dabei entstehenden Vektor und dem Sichtvektor berechnet. Dies wird mithilfe des Skalarproduktes kalkuliert. Zum Schluss werden Werte, die geringer als Null sind, entfernt und eine Potenz hinzugefügt, mit welcher in der Lichtquelle die spekulare Intensität individualisiert werden kann. Dies wird mit einer Spekulartextur , um eine beliebige Einfärbung zu ermöglichen. Der entstehende Vektor wird dann mit der diffusen Schattierung maskiert, sodass an unbeleuchteteten Stellen kein spekularer Punkt auftritt. Wenn Distanzabschwächung benötigt wird, wird diese noch mit dem Ergebnis dessen multipliziert.

9.3 Der Texturpass im Detail

Innerhalb des Texturpasses werden alle optischen Elemente berechnet, die keine Verbindung zur Beleuchtung haben. Dies umfasst die Grundfarbe und Reflektivität.

9.3.1 Blending

Um nun die Beleuchtungsmap weiter zu verarbeiten und gleichzeitig als Maske zu nutzen, muss diese multiplikativ mit dem Rückgabewert dieses Passes geblendet werden. So ergibt sich die gewünschte Einfärbung und unbeleuchtete Stellen verbleiben als solche.

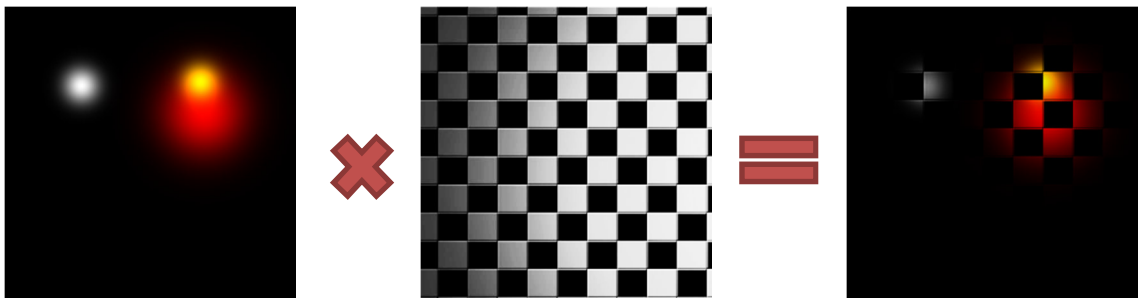


Abbildung 12: Die endgültige Beleuchtungstextur, wie sie auf einem dreidimensionalen Objekt zu finden wäre.

9.3.2 Der Vertexshader

Innerhalb des Vertexshaders wird die Position des Meshes neu berechnet und die Texturkoordinaten zwischengespeichert. Um Reflektionen anhand der Normalen zu ermöglichen, benötigt man weiterhin die Normalen-, Binormalen- und Tangentvektoren,

genauso wie den Sichtvektor der Kamera. Diese werden in einem entsprechendem Struct abgespeichert und dem Fragmentshader als Parameter übergeben.

9.3.3 Der Fragmentshader

Im Fragmentshader werden erneut die Normalentexturberechnungen durchgeführt. Diese werden benötigt, um eine berechnet Reflexion basierend auf auf den neuen Normalenausrichten zu bestimmen. Darüber hinaus wird die Diffustextur gelesen und mit einer optionalen Farbe multipliziert, um eine Einfärbung zu ermöglichen. Um Reflektionen zu erhalten, muss die Cubemap gelesen werden. Diese erhält als UV-Koordinaten den Sichtvektor, der an der Normalenrichtung reflektiert wird.

10 Die Performanz im Vergleich

Innerhalb der Unity3D Engine gibt es weitere Beleuchtungstechnologien, die alle unterschiedliche Vor- und Nachteile haben. Diese können, je nach den Projektanforderungen, auch miteinander kombiniert werden, um so die beste Performanz des Videospiels zu ermöglichen.

Die dynamische Beleuchtung ist das unperformanteste der von Unity eingebauten Systeme. Bei diesem werden die Lichtpositionen innerhalb des Shaders in jedes gerenderten Bildes aktualisiert, sodass dynamische Objekte auch dynamische Schatten werfen und gleichzeitig eine sich verändernde Beleuchtung aufweisen. Hierzu wird lediglich eine beliebige Lichtquelle und ein Mesh mit einem Shader, der diese Art der Beleuchtung unterstützt, benötigt. Jegliche Schritte darüber hinaus regelt die Engine intern.

Seit der Unity3D Version 3.0 hat man die Möglichkeit, auf Beast Lightmapping zuzugreifen. Dieses erstellt für jedes statische Objekt in der Szene eine Textur, welche die Beleuchtungsinformationen beinhaltet. Diese ersetzt dann jegliche Beleuchtungsberechnungen. Somit sind dynamische Elemente wie der spekulare Punkt oder sich ändernde Schatten nicht möglich.

In dem in dieser Arbeit thematisierten System kann man wählen, ob es als komplett dynamische Beleuchtung ohne Schatten oder als statische Lichtquelle mit der Fähigkeit zur Beleuchtung dynamischer Objekt genutzt werden kann.

Im Folgenden werden drei verschiedene Szenen betrachtet. Die erste besitzt 5200 gerenderte Dreiecke, die zweite 16700 und die dritte 70500. In jeder der Szenen werden die o.g. Systeme verwendet und die dazugehörigen Bilder pro Sekunde und Drawcalls gemessen, um klare Vergleichsgrößen für die Performanz des Shaders zu haben. Der speziell für dieses System angefertigte Shader untstützt drei Lichtquellen, welche alle das Phong-Beleuchtungsmodell nutzen. Für die Szenen mit der dynamischen Beleuchtung und dem Beast Lightmapping wurde ein vergleichbarer Shader geschrieben, der aber das interne Beleuchtungssystem von Unity nutzt. Um eine vergleichbare Qualität aufzuweisen, wurde die maximale Größe der Lightmaps gewählt, welche eine Auflösung von 1024 x 1024 definiert. Um den besseren Vergleich zu haben, wurde das Werfen von Schatten für diese beiden System deaktiviert. Für alle Szenen wurde nur ein einziges Material für alle Objekte verwendet, welches keine Texturen nutzt aber die entsprechenden Kalkulationen für die Standardwerte durchführt.

Im Anhang befindet sich das gesamte Projekt mit allen Testszenen, Texturen, Skripten, Shadern und Materialien.

Die folgenden Werte sind auf einem Alienware M14XR2 mit einer GeForce GT 650M und einem i7-3630QM mit 2,40 Ghz, genauso wie 12 GB Ram durchgeführt.

	Szene 1: 5200Tris	Szene 2 16700 tris	Szene 3 70500tris
Ohne Beleuchtung	Ø 47 FPS 2 DC	Ø 41 FPS 3 DC	Ø 39 FPS 5 DC
Dynamische Beleuchtung	Ø 40 FPS 13 DC	Ø 29 FPS 51 DC	Ø 31 FPS 103 DC
Beastlightmapping	Ø 46 FPS 2 DC	Ø 40 FPS 4 DC	Ø 38 FPS DC 8
Shaderbasierende Beleuchtung	Ø 34 FPS 5 DC	Ø 25 FPS 9 DC	Ø 22 FPS 17 DC

Tabelle 3: Messwerte der verschiedenen Szenen in Frames per Second (FPS) und Drawcalls (DC).

Im Hinblick auf die Messwerte ist klar zu erkennen, dass das Beast Lightmapping unabhängig von der Szenenkomplexität die performateste Technologie repräsentiert. Dies wird sowohl im Hinblick auf die Bilder pro Sekunde als auch auf die Drawcalls, bestätigt. Für statische Objekte ist dies in jeder Hinsicht die Beste der drei Lösungen. Dennoch ist diese Technologie für dynamische Umgebungsberechnungen nicht geeignet.

Das interne dynamische Beleuchtungssystem von der Unity3D Engine ermöglicht bis zu 9 Bilder pro Sekunde mehr als die in dieser Arbeit erstellten Technologie. Positiv ist dennoch zu vermerken, dass die Anzahl der Drawcalls im geringsten Fall um acht und im höchsten Fall um 86 geringer sind. Somit würde das System besonders umfangreiche Szenen durch einen großen Shader, der sehr viele Berechnungen besitzt, gerade im Bereich Drawcalls stark entlasten und trotzdem dynamische Beleuchtung ermöglichen.

11 Die Abschlussbetrachtung

Die für diese Arbeit erstellte Technologie erweist sich im Vergleich zu den bereits Vorhandenen als leichter zu modifizieren und zu erweitern, da alle Berechnungen offen zugänglich sind. Man benötigt keine besonderen Zugriffsrechte oder Lizenzen, alle Individualisierungen können mithilfe eines Shaders durchgeführt werden. Dieser kann die Berechnungen der Lichtpässe halten und diese jederzeit für neue Shader abrufbar machen. So ermöglicht es Unity, durch das Nutzen der UsePass-Funktion[44] innerhalb des Shaders, diese Pässe intern in den Shader zu kopieren und macht somit dieses System flexibel für jegliche, gebräuchliche Beleuchtungsshader. Weiterhin verringert es die Restriktionen der Indie Version der Engine, was gerade für private Projekte nützlich sein kann. Da es eine dynamische Beleuchtung mit unendlich vielen Lichtquellen ermöglicht, sind realistischere Lichteffekte innerhalb von Szenen möglich.

Dennoch ist es im Hinblick auf die Performanz weniger geeignet als die internen Systeme der Unity3D Engine und somit nur für Projekte geeignet, die starke Individualisierung in diesem Aspekt benötigen. Grund dafür ist die Masse an Berechnungen, die nun anstelle von C#, vom Shader übernommen werden. Der Faktor, dass diese sogar im Fragmentshader durchgeführt werden, um eine bessere optische Präsenz zu ermöglichen, erhöht den Aufwand weiterhin.

Den Nachteil, dass man keine Schattenberechnung verwenden kann, ist nur mit der professionellen Version der Unity3D Engine zu ändern. In dieser hat man Zugriff auf den Tiefenbuffer und dadurch wird ermöglicht, im Zusammenspiel mit Raycasts, Schattentexture zu erstellen.

Für eine Weiterentwicklung gibt es mehrere Aspekte, die man für dieses System in Betracht ziehen könnte. Zuerst wäre es für Entwickler ohne Kenntnisse von Programmierung, die dieses System nutzen von Vorteil, einen Shaderwizzard nutzen zu können, um die gewünschten Shader mit wenigen Klicks zu individualisieren und damit nur noch benötigte Features im Shader zu haben. Das würde eine Verbesserung der Performanz ermöglichen. Weiterhin wäre es vorteilhaft die Option zu haben, die Materialien und Lichtquellen einer mit dem internen Beleuchtungssystem von der Unity3D Engine fertig gestellte Szene analysieren zu lassen und diese mit den

---

[44] Unity Technologies: ShaderLab syntax: UsePass. 2014. URL: http://docs.unity3d.com/Manual/SL-UsePass.html, letzter Zugriff: 27.01.2015

Elementen dieses Systems zu ersetzen. Damit wäre eine einfach Konvertierung möglich.

Abschließend ist festzustellen, dass die komplette Beleuchtungsrechnung nicht in einem Shader durchgeführt werden sollte, da die Performanz während der Laufzeit stark darunter leidet. Möglichst viele Schritte sollten von der CPU vorberechnet sein, um die Grafikkarte zu entlasten.

Literaturverzeichnis

Jon Peddie: The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR and AR. Springer Science & Business Media. London 2013

Henri Gouraud: Continuous Shading of Curved Surfaces, Juni 1971, URL : http://page.mi.fu-berlin.de/block/htw-lehre/wise2012_2013/bel_und_rend/skripte/gouraud1971.pdf. Letzter Zugriff 27.01.2015

Sheilly Padda et al : Different Shading Algorithms for Image Processing, Mai 2014. URL: http://www.ijarcsse.com/docs/papers/Volume_4/5_May2014/V4I5-0472.pdf, letzter Zugriff: 27.01.2015

Eric Haines et al: Real-Time-Rendering. AK Peters, Ltd. Februar 2012.

Kenny Lammers: Unity Shaders and Effects Cookbook. PACKT Publishing. Juni 2013

James F. Blinn : Models of Light Reflection For Computer Synthesized Pictures, Juli 1977. URL: http://research.microsoft.com/pubs/73852/p192-blinn.pdf , letzter Zugriff: 27.01.2015

Future plc: Assassin's Creed IV is being developed across seven different studios – here's how, URL: http://www.edge-online.com/news/assassins-creed-iv-is-being-developed-across-seven-different-studios-heres-how/#null. März 2013. letzter Zugriff: 27.01.2015

Nicodemus et al: Geometrical Considerations and Nomenclature for Reflectance. Oktober 1977. URL : http://graphics.stanford.edu/courses/cs448-05-winter/papers/nicodemus-brdf-nist.pdf, letzter Zugriff: 27.01.2015

Unity Technologies: ShaderLab syntax: SubShader Tags. 2014. http://docs.unity3d.com/Manual/SL-SubshaderTags.html, letzter Zugriff: 27.01.2015

Unity Technologies: Making multiple shader program variants. 2014. http://docs.unity3d.com/Manual/SL-MultipleProgramVariants.html , letzter Zugriff: 27.01.2015

Unity Technologies: Vertex and Fragment Shader Examples. 2014. http://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html , letzter Zugriff: 27.01.2015

Unity Technologies: ShaderLab syntax: Blending. 2014. http://docs.unity3d.com/Manual/SL-Blend.html, letzter Zugriff: 27.01.2015

Unity Technologies: ShaderLab syntax: SubShader Tags. 2014. http://docs.unity3d.com/Manual/SL-SubshaderTags.html, letzter Zugriff: 27.01.2015

Unity Technologies: ShaderLab syntax: Pass Tags. 2014. http://docs.unity3d.com/Manual/SL-PassTags.html , letzter Zugriff: 27.01.2015

Anlagen

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum Vorname Nachname